Method Popularity Distributions of Software Artefacts within Maven Central

Thijs Nulle
Supervisor: Mehdi Keshani
EEMCS, Delft University of Technology, The Netherlands
20-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Even though previous studies have studied software artefacts on a package level, little research has been done on a method level. In this work, we perform a method-level analysis to determine how popularity disperses among methods within software libraries of Maven Central. We analyse 384 software artefacts with three different metrics: eigenvector centrality, degree centrality and dependent usage percentage. Using callgraphs of the interactions of a software artefact with its dependents, we can determine the relative popularity score of any method. We observe that popularity is inverse logarithmically distributed among the most frequently used methods within a library. Furthermore, 80% of calls to a library are to 26% of all methods, following the Pareto Principle. Likewise, the number of dependents per artefacts also follows a power-law distribution. We also find that no significant correlation exists between any of the analysed metrics, allowing opportunities for future research to determine a more accurate popularity metric. All of our results show that method popularity is logarithmically distributed within software artefacts of Maven Central.

1 Introduction

Software libraries provide endpoints for reusing functionality within a software ecosystem. These endpoints are represented as APIs, *Application Programming Interfaces*, allowing users to integrate external software into their application effortlessly. The distribution of those library packages often occurs through a project management tool, e.g. a package manager. The project management tool analysed in this paper is *Maven Central* [1], an ecosystem designed for dependency management and standardising the build process of Java applications.

Benelallam et al. [2] present the *Maven Dependency Graph*, a dataset comprising 2.4M indexed artefacts of Maven Central, allowing one to analyse processes, trends and dependency relations within the ecosystem. Utilising the Maven Dependency Graph, Soto-Valero et al. [3] analysed artefacts based on their version usage, popularity and timeliness. Both of these papers perform a similar study, but they analyse at a package level, whereas we analyse at a method level.

In this paper, we pose the following research question: *How is popularity distributed among methods within a software library?* We present a procedure to determine how popularity disperses within a software artefact. By analysing how software artefacts interact with any library, we can discern what methods influence the ecosystem more than others.

For this study, we investigate the interactions between different software artefacts. To capture these interactions, we use the Fasten framework [4]; it resolves dependents and dependencies of any package allowing us to map dependency relations. Using these dependency relations, we can generate callgraphs to analyse; where a callgraph is a directed graph where the nodes represent a method M, where each edge represents a method M_i invoking another method M_j [5].

Prior to callgraph generation, we have to select a subset of artefacts that are representative of Maven Central. We perform weighted random sampling where the weight is the number of dependents an artefact has. We select 384 artefacts to analyse, achieving a confidence level of 95% with a margin of error of 5%.

Subsequently, we generate callgraphs of all interactions between any dependent and its dependencies; note that the artefact we are analysing is part of the dependencies. We analyse the callgraphs based on three metrics, explained briefly in the following paragraph.

The metrics we consider are *Eigenvector Centrality*, *Degree Centrality* and *Dependent Usage Percentage*. Eigenvector centrality determines the importance of a node, considering both the number of connections and the quality of the connection itself. Degree centrality is the number of edges, $|E|$, connecting to any node. Finally, dependent usage percentage is the percentage of dependents that call a given method M.

Our key findings, presented in Section 4, show that method popularity follows a logarithmic distribution for the most used methods. However, infrequently used methods do not have enough data to determine a popularity score. Consequently, no correlation exists between eigenvector centrality, degree centrality and dependent usage percentage; a method can have a high score in one metric and a low score in another. Finally, we determined that method calls and the number of dependents of any artefact follow the Pareto Principle, which states that for many phenomena, 80% of the consequences are caused by 20% of the causes [6]. On average 26% of all methods account for 80% of all method calls within any software artefact.

This paper is structured as follows. In Section 2, we define a list of definitions, followed by a list of related works. Section 3 contains information about the methodology to conducting our research. It briefly states an overview of the approach, followed by an explanation of the research question. Subsequently, we cover the data specification and methodology. In Section 4, we show and explain the results. Finally, we discuss our findings in Section 5.

We analyse 384 software artefacts of Maven Central to determine a popularity score for several metrics; eigenvector centrality, degree centrality and dependent usage percentage. We establish an approach to determine a popularity ranking for any software artefact. Our main contributions are as follows:

- comprehensive analysis of method popularity within Maven Central
- dataset of 384 analysed artefacts, including callgraphs and popularity values
- tools for generating and analysing callgraphs

2 Background

In the following section, we first explain the definitions used in this paper, followed by a brief overview of the related works preceding this paper.

2.1 Definition of Terms

Maven Central[1] is an automated build tool used mainly for Java projects. One can specify dependencies on other software artefacts within a `pom.xml` file.

In this paper, we consider an *artefact* to be a software artefact with a unique $groupId$, $artifactId$ and $version$ in the following format: `groupId:artifactId:version`.

A *dependency* another software artefact our current project needs to compile, build, test or run.

A *dependent* is the contrary of a dependency; for any given artefact A_A if they specify another artefact A_B in their list of dependencies, A_A is a dependent of A_B.

Dependency resolution is resolving what dependencies a package needs to compile, build, test or run; specified in the `pom.xml` file. For more accurate dependency resolution, one recursively needs to get the dependencies of all dependencies.

Dependent resolution is resolving what other packages depend on your package; one needs to perform dependency resolution for all packages. Using this information, one can create a so-called dependency graph and determine, using the in-degree of a node, which packages are dependent on your package.

A *callgraph* is a directed graph where the nodes represent a method M, where each edge represents a method M_i invoking another method M_j [5].

Fasten is an intelligent software package management system. Using Fasten, one can resolve dependencies and dependents for a software package [4]. Using this information, Fasten can generate callgraphs of the interactions between these software packages, allowing one to analyse the interdependence relations.

2.2 Related Works

In the following section, we review the existing studies in the literature. Firstly, we cover two papers that present datasets to analyse Maven Central, the *Maven Dependency Dataset (MDD)* and the *Maven Dependency Graph (MDG)*. Subsequently, we look at a paper that uses the MDG to analyse software diversity within Maven Central. And finally, we cover an article which investigates a seeming paradox for API usage.

Raemaekers et al. [7] present the Maven Dependency Dataset, which contains metrics, changes and dependencies of various JAR files. The results are on the level of individual methods, classes and packages of multiple library versions. The dataset includes several centrality measures, for instance page rank and betweenness, and rudimentary usage frequencies on a method level.

Benelallam et al. [2] present the Maven Dependency Graph, containing metrics, changes and dependencies of a large number of JAR files within Maven Central; it consists of 2.4M indexed artefacts and 9M dependency relations. Their second contribution is providing procedures to query information from this dataset, such as artefact retrieval in time or per version range. With these two contributions, one can answer high-level research questions about artefact releases, evolution and usage trends over time.

[1]For more information, see `maven.apache.org`

Using the Maven Dependency Graph, Zerouali et al. [8] performed an empirical study on the emergence of software diversity within Maven Central. They extended the dataset to include dependency relationships at the library level to allow for a more in-depth analysis. They measure activity, popularity and timeliness between different artefact versions of a subset of the Maven Dependency Graph. Their contributions include a quantitive analysis of popularity between different library versions.

Harrand et al. [9] investigate a seeming paradox between Hyrum's law and the observation that most client dependencies focus on a small part of APIs. Hyrum's law states that given enough users, one will depend on any observable behaviour of your system [10]. These seemingly contradictory statements balance each other; a small part of the exposed API is most used, even though some libraries depend on the most uncommon behaviours.

Considering the MDD and the MDG, we concluded they either performed no or a rudimentary method level analysis regarding popularity, leaving an opportunity for researching the given topic. As Benelallam et al. [2] studied popularity between different versions of an artefact, it allows for extending this research at a method level. Finally, Harrand et al. [9] states in their study that most usages come from a small number of methods, compared to most methods having infrequent use. All previous statements confirm an opportunity for in-depth research regarding method popularity within software artefacts.

3 Approach

This section starts with an overview of the approach to ensure the subsections are comprehensible in the given order. Subsequently, we will elaborate on the research question, describe the data selection process and explain our methodology.

3.1 Overview of Approach

Figure 1 illustrates an overview of the approach outlined in this section. The first step in the approach is selecting a subset of artefacts to analyse within Maven Central. Subsequently, we resolve a list of dependents for all artefacts. For the dependents of any artefacts, we resolve their dependencies, containing the original artefact.

Afterwards, we generate a callgraph of all interactions between any dependent and its dependencies. We aggregate these generated callgraphs into one combined callgraph. Finally, we analyse this combined callgraph based on degree centrality, eigenvector centrality and dependent usage.

RQ. How is popularity distributed among methods within a software library? We aim to propose a procedure for determining how popularity disperses among software methods. We analyse several software packages' dependents and investigate what methods they use and how consequential those methods are within the given software package.

3.2 Data Specification

To analyse a large dataset like the Maven Central repository, one needs to select a representative subset. Taherdoost [11]

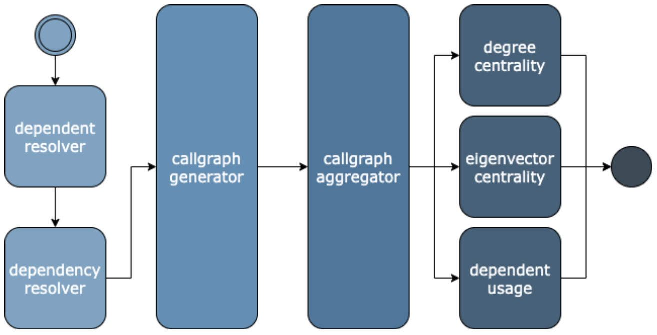

Figure 1: Application Flow of the Approach

proposes a list of stages for a sampling process, including several sampling techniques with their advantages and disadvantages. For a concise overview of the sampling process, see the end of this subsection.

Sampling Techniques Before we can sample any data, we need to determine which sampling technique to use. We considered weighted random sampling and stratified random sampling. Stratified random sampling is where before the sampling process, we split the dataset into subgroups and samples are selected from those. Ultimately, we opt for the weighted random sampling technique as no metric exists to divide the dataset fairly.

Sampling Frame To select a sampling frame, we need to define a target population, which in our case is the Maven Central repository, consisting of approximately 9M indexed artefacts. Fasten contains the most recently added artefacts, and thus we opt for all artefacts added between the 1st of October 2021 and the 31st of March 2022 as our sampling frame. The final sampling frame contains approximately 400K artefacts. See Figure 2 for the number of artefacts loaded in Fasten within the sampling frame.

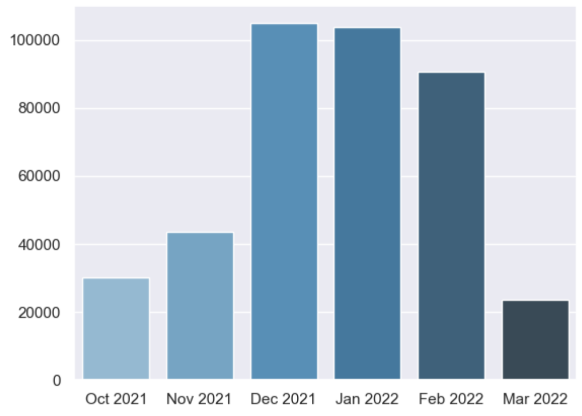

Figure 2: Artefacts added per month within the sampling frame

Pre-processing Artefacts Before we sample our dataset, we apply two filters to the sampling frame; which allows us to create a more representative dataset.

The first filter we apply is to remove testing-related[2] artefacts from the dataset. As all artefacts uploaded to Maven Central only contain source code, there is no feasible way to analyse any testing-related artefacts. After applying the filter, approximately 380K artefacts remain.

Subsequently, we filter multiple versions of any artefact. Since different versions of the same artefact likely will see the same usage pattern, it is more relevant to analyse different artefacts compared to different versions. After applying the filter, approximately 10K artefacts remain.

Data Selection After filtering the sample frame, we consider the steps of sampling the dataset. As artefacts with more dependent relations influence the ecosystem more, we perform weighted random sampling based on the number of dependents of any artefact, see Table 1 for a distribution of the dependents count.

Num Dependents	Num Artefacts
0	7533
1	2091
2-9	844
10-24	88
25-49	33
50+	27

Table 1: Distribution of number of dependents per artefact

Within a list of dependents of an artefact, one might find several dependents with the same $groupId$ as the artefact itself; more elaborate artefacts subdivide into more concise artefacts with mutual dependencies. We opt to keep only one dependent per unique $groupId$, ensuring the data selection process does not get skewed by these dependence relations.

After resolving dependents for all pre-processed artefacts, roughly 7500 artefacts do not have any dependents, which using weighted random sampling cannot be selected. Based on a maximum margin of error and a confidence level, Taherdoost [12] proposes a method to determine the sample size. Based on the tables presented in the paper, we choose a sample size of $n = 384$, achieving a confidence level of 95% and a maximum margin of error of 5%. For the 10 selected artefacts with the most dependents, see Table 2.

Overview of Sampling Process

1. Select all Maven artefacts between October 1st 2021, and March 31st 2022
2. Filter all testing-related artefacts
3. Filter artefacts based on a unique version
4. Calculate the number of dependents for every artefact
5. Perform weighted random sampling with the weight being the number of dependents

[2]We filter all the artefacts that contain the following words: `assertj`, `junit`, `mock` and `test`.

Artefact Name	Number of Dependents
com.google.code.gson:gson:2.8.9	539
ch.qos.logback:logback-core:1.2.7	328
org.projectlombok:lombok:1.18.22	244
com.fasterxml.jackson.core:jackson-annotations:2.12.6	228
org.bouncycastle:bcprov-jdk15on:1.70	228
org.yaml:snakeyaml:1.30	174
joda-time:joda-time:2.10.13	133
com.squareup.okhttp3:okhttp:4.9.2	117
com.fasterxml.woodstox:woodstox-core:6.2.7	110
com.github.ben-manes.caffeine:caffeine:2.9.3	104

Table 2: Top 10 Analysed Artefacts with the most Dependents

6. Sample 384 artefacts to achieve a confidence level of 95% and a margin of error of 5%

3.3 Methodology

Our goal is to determine a popularity distribution of a method within a software package of Maven Central. In the following section, we propose the steps needed to accomplish this goal.

Generating Callgraphs To analyse a software artefact, one needs to generate a callgraph containing all the interactions of the given artefact and its dependents. The algorithm to generate such a callgraph is given in Algorithm 1.

Algorithm 1 Callgraph Generation Algorithm

```
1: function GENERATEJOINEDCALLGRAPH($P$)
2:     $Dep_P \leftarrow resolveDependents(P)$
3:     for all $d_i \in \mathcal{D}ep_P$ do
4:         $Dpy_{d_i} \leftarrow resolveDependencies(d_i)$
5:         $CG_i \leftarrow generateCallgraph(Dpy_{d_i})$
6:     end for
7:     $CG_P \leftarrow joinCallgraphs(CG)$
8: end function
```

Using the list of artefacts as inputs, we generate a callgraph for every artefact and its dependents. As the first step of the algorithm, we resolve the list of dependents for all artefacts; we determine which specific versions of any artefact declare a dependency on the target artefact.

Using this list of dependents, we need to determine how every dependent interacts with our artefact. To determine all method calls from a dependent to a dependency, one needs to generate a callgraph that includes them and their dependencies. Note that the dependencies of a dependent contain the artefact we are analysing.

Because of inheritance in object-oriented programming, one needs to perform a so-called *class hierarchy analysis* to ensure all possible method calls are contained within the callgraph. A class hierarchy analysis determines a program's class inheritance graph and the set of methods defined on each class [13]. Using these two pieces of information, we can add all possible invocations of any method to the callgraph.

As the final step of creating a callgraph of a software artefact, we need to join all the partial callgraphs. Because all methods within Fasten have unique identifiers, we can combine all edges of the callgraphs using Equation 1. See Figure 3 for a visual representation of joining callgraphs.

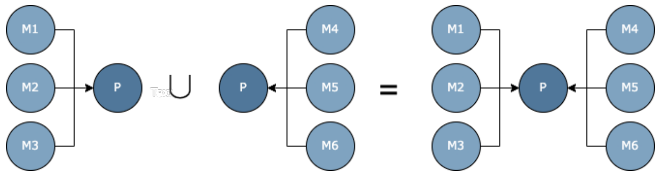

Figure 3: Example of callgraph joining

$$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(a,b) :\in V_1, b \in V_2\}) \quad (1)$$

where:

G_x = Graph x
V_x = Vertices of graph x
E_x = Edges of graph x

After executing this algorithm, we have the most accurate representation of the interactions between an artefact and its dependents, all in a single callgraph. By analysing the callgraph of an artefact, we can determine popularity scores based on all interactions with that artefact.

Metrics The first step of the analysis is to devise a list of metrics we use to analyse the data. As the data consists of directed graphs, we consider several proposed metrics within graph theory. In the following paragraphs, we cover *Eigenvector Centrality*, *Degree Centrality*. Finally, we also cover a simple metric comparing the percentage of dependents that call a given method.

The first metric we use is *Eigenvector Centrality*, which is a measure that takes into consideration both the number and the quality of the connections between nodes [14]. See Equation 2 for the mathematical formula. Since we want to determine the popularity of a given method M_i, it is important to consider the relative importance of a method M_j that calls the method.

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t \quad (2)$$

where:

x_v = relative centrality score of v
λ = constant, eigenvalue in the vector notation $Ax = \lambda x$
$M(v)$ = set of neighbours of v

The second metric we use is *Degree Centrality*, defined as the number of edges, $|E|$, that connect to a given vertex V within a graph $G := (V, E)$. See Equation 3 for the mathematical formula. As we use degree centrality for a directed graph, we need to differentiate between indegree and outdegree. We use indegree since we care about methods to our artefact, represented as an edge from a method M_i to another method M_j.

$$C_D(v) = deg(v) \quad (3)$$

where:

$C_D(v)$ = degree centrality of v
$deg(v)$ = outgoing edges of v

The final metric is the percentage of dependents that call a given method M. See Equation 4 for the mathematical formula. As this metric cannot be influenced by multiple method calls from the same dependent, it might give different insights into method popularity.

$$Dep_{\%}(m) = \frac{1}{n} \sum_{d \in Dep_m} \begin{cases} 1 & \exists\{d, m\} \in CG_P \\ 0 & otherwise \end{cases} \tag{4}$$

where:

$Dep_{\%}(m)$	= percentage of dependents that call a method m
Dep_m	= dependents set of a method m
$\{d, m\}$	= edge between a dependent d and a method m
CG_P	= callgraph of a package P

4 Results

In the following section, we will show the results of the approach described in Section 3. Firstly, we will briefly go over the research question and the metrics. Subsequently, we will show and explain the figures for the popularity distributions.

In this paper, we aimed to answer the following research question; *How is popularity distributed among methods within a software library?*. To answer this question, we proposed the following metrics; eigenvector centrality (see Equation 2), degree centrality (see Equation 3) and dependent usage percentage (see Equation 4).

4.1 Metrics

In this section, we cover the three metrics proposed in Section 3.3. We explain the popularity distribution of the given metric, including discrepancies, if applicable. Firstly, we cover eigenvector centrality, followed by degree centrality and dependent usage percentage.

Eigenvector Centrality As seen in Figure 4, the data follows a logarithmic distribution within the first four quintiles; a small number of methods are of great significance compared to their peers. In the last quintile, the distribution contains a drop-off because we aim to determine a popularity value for infrequently called methods, resulting in high variance.

Degree Centrality As seen in Figure 5, degree centrality follows a logarithmic distribution with more emphasis on the popular methods. With eigenvector centrality, all given method calls are aggregated into a single value, whereas all individual method calls contribute to degree centrality, amplifying the differences in the distributions.

Dependent Usage Percentage As seen in Figure 6, dependent usage percentage follows a stronger logarithmic distribution compared to degree centrality. Because dependent usage percentage limits the number of calls from a dependent to a method to one and the values between artefacts are not normalised, the distribution is more steep than the degree centrality distribution; most methods are invoked in a small percentage of dependents.

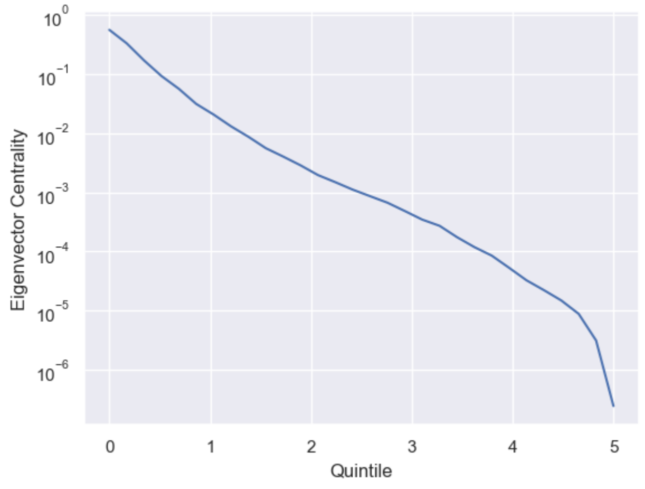

Figure 4: Popularity Distribution for Eigenvector Centrality

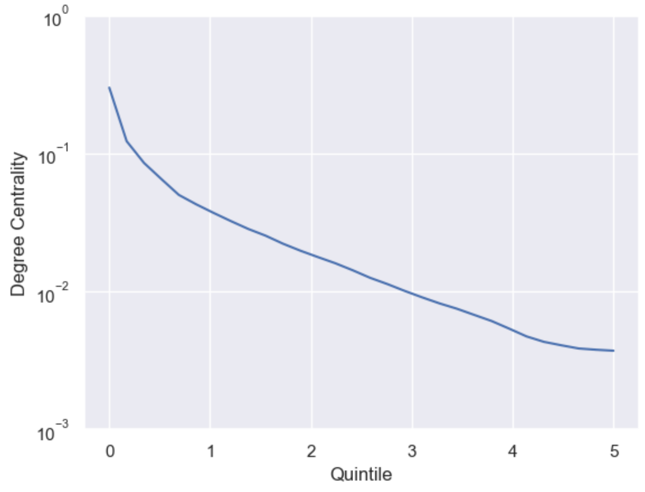

Figure 5: Popularity Distribution for Degree Centrality

Influence of Internal Usage A possibility we considered was the influence of non-unique $groupId$ dependents on the results; larger artefacts are subdivided into smaller artefacts with the same $groupId$ and mutual dependencies. To ensure this did not influence the results, we ran the experiments while filtering all dependents with the same $groupId$.

Finally, the results did not change significantly; it still follows the same distribution. See Figure 7 for the popularity distribution of eigenvector centrality with the unique $groupId$ filter; the distributions for the other metrics are omitted.

4.2 Metric Correlations

As all distributions follow a logarithmic distribution, it is no surprise that one can use any of the proposed metrics to calculate method popularity within software artefacts. The similarity between the distributions may be that all three metrics consider the same kind of connections. In the following

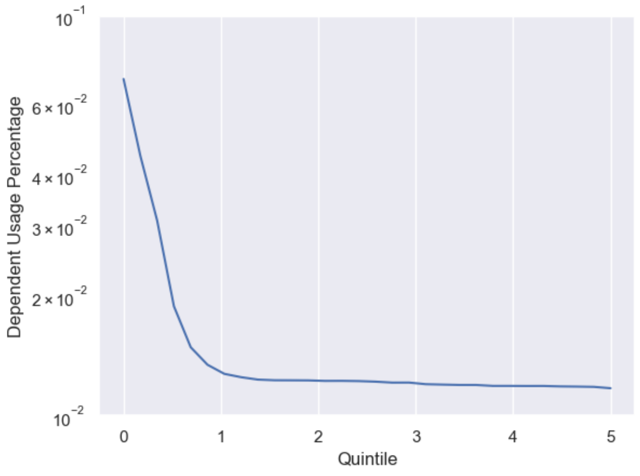

Figure 6: Popularity Distribution for Dependent Usage Percentage

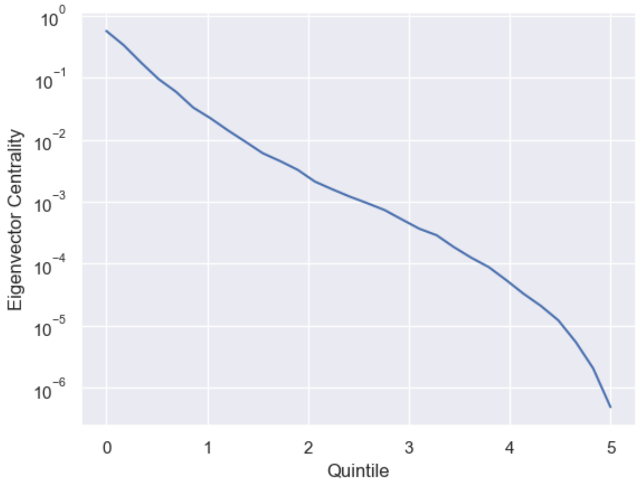

Figure 7: Popularity Distribution for Eigenvector Centrality with Unique $groupId$ Dependents

paragraphs, we will elaborate on the correlations between the metrics utilising 2-dimensional plots, with both axes being a metric.

Firstly, we cover a possible correlation between eigenvector centrality and degree centrality, as shown in Figure 8. It seems no correlation exists between eigenvector centrality and degree centrality within method usage. As highlighted by the distinct vertical line at $x = 10^0$, one can see that there are methods within software artefacts that are important when considering eigenvector centrality, but their usage is infrequent.

Subsequently, there exists a correlation between degree centrality and dependent usage percentage. As seen in Figure 9, a subtle trend exists from low values to high values for both metrics. There are several horizontal lines due to the limit of the number of dependents, as explained in the previous section. One can explain the correlation between the two metrics because dependent usage percentage is a more specific version of degree centrality, limiting the maximum number of calls to a method per artefact to one.

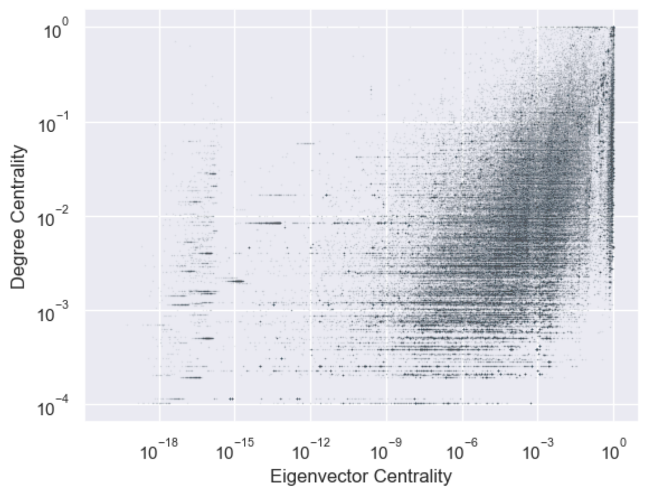

Figure 8: 2-Dimensional Popularity Distribution of Eigenvector Centrality and Degree Centrality

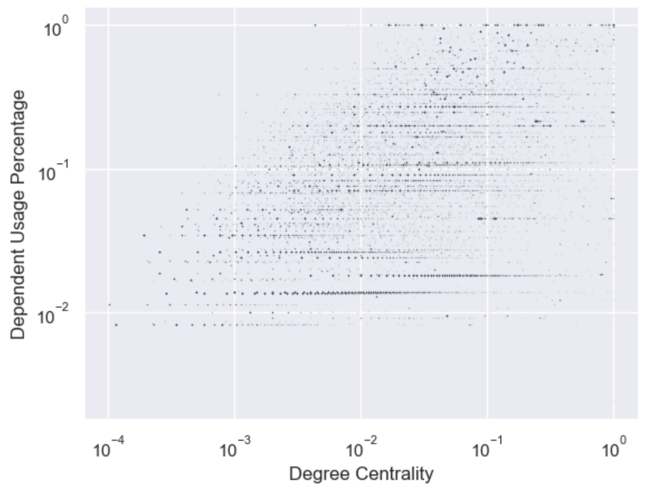

Figure 9: 2-Dimensional Popularity Distribution of Degree Centrality and Dependent Usage Percentage

5 Discussion

In this section, we firstly elaborate on the results of Section 4. Subsequently, we cover the implications of the results, followed by their uncertainties. We will cover some ideas for future research and possible integrations of our approach within software tooling. We also cover the threats to the validity of the study. Finally, we discuss the reproducibility and the ethical implications of our research in Section 5.1.

Interpretations The results indicate that there does exist a trend between different artefacts related to method popularity; they all follow a logarithmic distribution. The distributions also follow the *Pareto principle*, which states that roughly 80% of consequences come from 20% of the causes [6]. If we were to relate this to all callgraphs we generated, we established that approximately 80% of method calls are to 26% of the total methods. Subsequently, the number of dependents per artefact also follows a power law distribution, as shown in Figure 10.

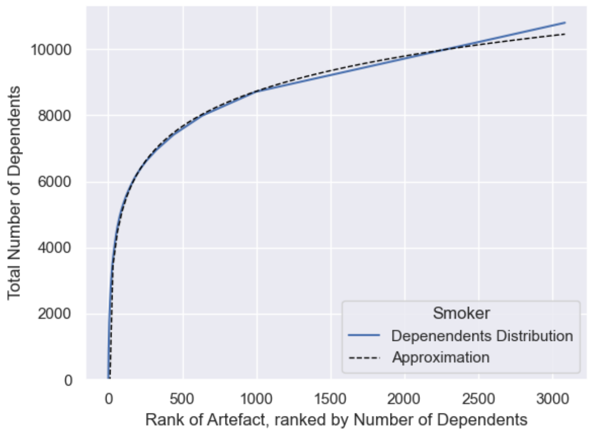

Figure 10: Distribution of Number of Dependents for all Artefacts

The statement regarding the Pareto Principle relates to the conclusions made by Harrand et al. [9], which state that most clients depend on a small fraction of an API and that, given enough users, people depend on all functionalities of an API. From our research, one can come to the same conclusion; popularity skews towards the most frequently used methods, but comparatively, more methods exist with low popularity values.

As all three proposed metrics only consider the method calls from a dependent to an artefact, albeit in different forms, they likely follow similar distributions as a consequence. If one looks at the intricate interactions between software artefacts, it might not give the complete picture to look at these metrics; we cannot ensure these are enough to determine an accurate popularity ranking.

When considering the boxenplots of the popularity distributions in Figure 11, one can see that the density of all metrics disperses differently among all analysed metrics. The middle quartile of eigenvector centrality resides among the higher values of the distribution, where with degree centrality and dependent usage percentage the middle quartile resides in the lower range of values. Even though the centre quartile shifts position between different metrics, for all metrics, only a several very popular methods exist.

Implications With an established popularity ranking, maintainers and users of a library can improve their workflow. Utilising such a popularity ranking, one could prioritise a list of issues to work on based on their importance within the ecosystem; a change to a popular method will have more

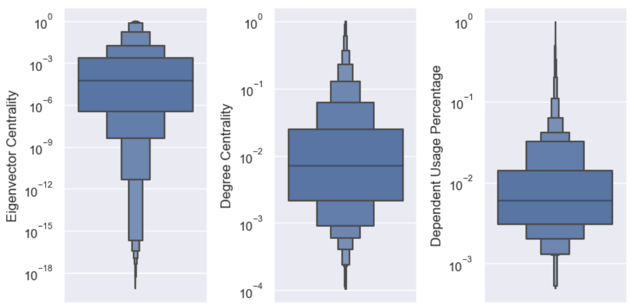

Figure 11: Boxenplot of the Popularity Distributions

impact compared to a lesser one. Likewise, an indecisive user could determine what method to use based on the popularity score of methods with similar functionalities.

Limitations A possible limitation with a popularity distribution among methods is the lack of data points with rarely used methods. In graph-based analysis, the more nodes and edges exist within a graph, the better the approximation becomes of the actual popularity value of a method. In essence, for rarely used methods, we are establishing a popularity score based on connections it does not have, which might skew the outcome incorrectly.

Future Research As the topic of method popularity within software ecosystems is unexplored, there are several opportunities for further research. We will cover three possible opportunities for future research in the following section.

The first research opportunity is regarding the metrics we used to determine the popularity of any method. As we analysed callgraphs, we opted for using graph-based metrics, whereas a more appropriate metric can be proposed to more accurately estimate popularity.

Consequently, one can use our findings and combine them with any other software metric to create a ranking of method importance within a software ecosystem. Using Fasten, one can analyse code based on quality and vulnerabilities, which both combined can reach more meaningful conclusions.

Finally, as we analysed the popularity distribution among methods, the next step is to determine why a method is popular. The most rudimentary approach is analysing if there exists any correlation between a method declaration and its popularity. Besides this, one could analyse if a correlation exists with a method's length or its cyclomatic complexity.

Future Work There are several applications where a popularity distribution can improve or add something to the workflow of a user or developer. We will cover one application for library developers and two for library users.

Firstly, developers can use a popularity ranking to make their workflow more efficient and streamlined. As any software project is an iterative process, there are always parts of a codebase that need improvement. A popularity distribution can give insight on how to prioritise work based on how much influence a method has on the users.

Secondly, whenever a user decides which library they will use for their project, they might compare the general popularity of the complete library. If a popularity overview was available for libraries with similar purposes, users could decide which library to use based on the method's popularity instead of the library's popularity. Also, a user could choose which method to utilise if there exist methods with similar functionality within one software artefact.

Finally, a popularity ranking could be integrated within an autocomplete engine. Whenever a user declares a dependency on a library, it might be hard to determine what they are trying to achieve with the library. However, using a popularity distribution, an autocomplete engine could suggest methods more accurately.

Threats to Validity As we performed many steps in this research, there are several threats to the validity of the results. In the following paragraphs, we will highlight and elaborate on them.

Firstly, Maven Central does not only contain Java artefacts; it also contains Scala and Kotlin artefacts. In some cases of these languages no callgraphs are associated with the artefacts, and thus the artefact itself is actually not analysed. While checking the data, we found that no more than 1% of analysed artefacts were Scala or Kotlin packages, but this could have more influence when reproducing the results.

Subsequently, there exists a significant difference in the number of methods between artefacts. Some artefacts have severals thousands of methods, whereas smaller libraries have as little as 10 methods. For the artefacts with a lower number of methods it might be possible that one cannot adequately determine a popularity distribution.

As the target population of our research contains 9M indexed artefacts, the sampling process may include mistakes in ensuring representativeness. Several possible issues may be; performing weighted random sampling where the weight is the number of dependents, filtering testing-related artefacts using keywords only and choosing a broader or narrower sampling frame.

We only consider invoked methods during callgraph generation; unutilised methods are not part of the generated callgraphs. As we determine a popularity score based on method calls, we cannot establish it for such methods. Thus, we cannot completely ensure the representativeness of the popularity distributions compared to the actual distributions.

During this study, we developed several programs which all possibly contain implementation errors. We can never ensure all implementations are bug-free, but performing rudimentary tests showed correct results for the selected samples.

5.1 Responsible Research

Responsible research and application of science and technology is supposed to foster dialogue in a global context and research on ethics of science and technology [15]. In the following section, we describe the reproducibility of our research, followed by what ethical implications our results might have.

Reproducible Package To ensure one can reproduce the results from our research, we present a reproducible package. Stodden et al. [16] states that having access to the computational steps of processing the data and generating findings is as important as access to the data themselves. The reproducible package is hosted on Docker Hub[3] and the source code is available on GitHub[4].

To reproduce the results, we provided two docker images to perform the callgraph generation and run the analysis scripts. As Fasten is still in development and no public API is available, we included the callgraph data and popularity values in the reproducible package. For more information about the reproducible package, including instructions on how to run the programs, see the `README.md` included in the reproducible package.

Ethical Implications When discovering possible ethical implications, the first place to look is at the possible usages of the research. The first usage is that developers could prioritise their workflow based on the popularity metrics of certain methods. The ethical issue associated with this is that, if hypothetically developers would adopt our popularity ranking, it might be biased to certain callgraph layouts or certain artefacts have more influence on the popularity value of a method. Both outcomes would change a developer's workflow such that they would not prefer.

Secondly, if a method popularity ranking would advise unknowing users on what methods to use, it might significantly influence the popularity of diminutive and or unpopular artefacts. It could slow the progress of an artefact's growth or not even let it become popular entirely.

6 Conclusion

While several research papers cover establishing a software library's popularity, we propose a method of determining a popularity distribution among methods within a software library. For a representative dataset of *Maven Central*, we generated callgraphs to analyse the interactions with these software artefacts. Using *eigenvector centrality*, *degree centrality* and *dependent usage percentage* we aimed to give meaning to these interactions in the form of a popularity score for each metric.

These findings show that the popularity distribution is logarithmic for all metricsKewar. The scores from all metrics follow a similar distribution, indicating that all metrics exploit some similar property of the methods. With this, one can determine what method of any software library is more popular than another, a rudimentary metric to determine importance within an artefact.

By analysing all interactions between software artefacts, we found that, on average, 26% of all methods account for 80% of all method calls within a software artefact. Subsequently, the number of dependents per artefact also follow a power law distribution.

Using the approach described in this paper, one could combine the findings with any other metrics to compare software artefacts. These combined metrics can contribute to more meaningful conclusions regarding popularity and approximate general importance within a software artefact. Subse-

[3]https://hub.docker.com/repository/docker/tnulle/maven-api-study

[4]https://github.com/thijsnulle/maven-api-study

quently, one could propose a metric tailored more towards method popularity compared with the graph-based metrics.

References

[1] Apache Software Foundation. *Apache Maven*. Version 3.8.5. URL: https://maven.apache.org/ (cit. on p. 1).

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. "The maven dependency graph: a temporal graph-based representation of maven central". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 344–348 (cit. on pp. 1, 2).

[3] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. "The emergence of software diversity in maven central". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 333–343 (cit. on p. 1).

[4] Fasten Project. *Fasten*. https://github.com/fasten-project/fasten. 2022 (cit. on pp. 1, 2).

[5] Barbara G Ryder. "Constructing the call graph of a program". In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 216–226 (cit. on pp. 1, 2).

[6] Rosie Dunford, Quanrong Su, and Ekraj Tamang. "The pareto principle". In: (2014) (cit. on pp. 1, 7).

[7] Steven Raemaekers, Arie van Deursen, and Joost Visser. "The Maven repository dataset of metrics, changes, and dependencies". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 221–224. DOI: 10.1109/MSR.2013.6624031 (cit. on p. 2).

[8] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. "On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 589–593. DOI: 10.1109/SANER.2019.8667997 (cit. on p. 2).

[9] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. "API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages". In: *Journal of Systems and Software* 184 (2022), p. 111134 (cit. on pp. 2, 7).

[10] Hyrum. *Hyrum's law*. URL: https://www.hyrumslaw.com/ (cit. on p. 2).

[11] Hamed Taherdoost. "Sampling methods in research methodology; how to choose a sampling technique for research". In: *How to Choose a Sampling Technique for Research (April 10, 2016)* (2016) (cit. on p. 2).

[12] Hamed Taherdoost. "Determining sample size; how to calculate survey sample size". In: *International Journal of Economics and Management Systems* 2 (2017) (cit. on p. 3).

[13] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of object-oriented programs using static class hierarchy analysis". In: *European Conference on Object-Oriented Programming*. Springer. 1995, pp. 77–101 (cit. on p. 4).

[14] Mark EJ Newman. "The mathematics of networks". In: *The new palgrave encyclopedia of economics* 2.2008 (2008), pp. 1–12 (cit. on p. 4).

[15] Mirjam Burget, Emanuele Bardone, and Margus Pedaste. "Definitions and conceptual dimensions of responsible research and innovation: A literature review". In: *Science and engineering ethics* 23.1 (2017), pp. 1–19 (cit. on p. 8).

[16] Victoria Stodden, Marcia McNutt, David H Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A Heroux, John PA Ioannidis, and Michela Taufer. "Enhancing reproducibility for computational methods". In: *Science* 354.6317 (2016), pp. 1240–1241 (cit. on p. 8).